An FPGA Implementation of the Powering Function with Single Precision Floating-Point Arithmetic*

Pedro Echeverría, Marisa López-Vallejo
Department of Electronic Engineering, Universidad Politécnica de Madrid (Spain)
Email: {petxebe, marisa}@die.upm.es

Abstract

In this work we present an FPGA implementation of a single-precision floating-point arithmetic powering unit. Our powering unit is based on an indirect method that transforms x^y into a chain of operations involving a logarithm, a multiplication, an exponential function and dedicated logic for the case of a negative base. This approach allows to use the full input range for the base and exponent without limiting the range of the exponent as in direct methods. A tailored hardware implementation is exploited to increase the accuracy of the unit reducing the relative errors of the operations while high performance is obtained taking advantage of the FPGA capabilities for parallel architectures. A careful design of the pipeline stages of the involved operators allows a clock cycle of 201.3 MHz on a Xilinx Virtex-4 FPGA.

1 Introduction

With the speed and size of current digital circuits increasing at the rate of Moore's law, FPGAs enable more developers to implement very sophisticated algorithms traditionally done in software running on slower general purpose processors. Moreover, FPGAs are specially characterized by their flexibility at a suitable low cost. For these reasons, FPGAs are now expanding their traditional prototyping roles to help offload computationally intensive functions from the processor.

On the other hand, many of these computationally intensive applications that are good candidates for hardware acceleration require high precision, floating point (f.p.) arithmetic. In this context, developing f.p. designs on FPGAs is still challenging, because the hardware implementation of f.p. operators requires very large areas and extremely deep pipelines.

There are numerous previous works on the implementation of f.p. arithmetic operators on FPGAs [1, 2]. Nevertheless, they are mainly focused on the four basic mathematical operators: Addition/Subtraction, Multiplication, Division and Square Root.

Other operators have not been studied so deeply, basically due to their enormous complexity. Among these operators we can find the trigonometric functions or other complex arithmetic operators as the exponential function or the logarithm. This is the case of the powering function x^y, implemented in this work, a computationally intensive function widely used in scientific computing, computer 3D graphics or signal processing. Due to the its complexity, the

*This work has been partly funded by BBVA under contract P060920579 and by the Spanish Ministry of Education and Science through the project TEC2006-00739.

direct implementation in hardware of the powering function unit with single f.p. arithmetic is not feasible. In previous literature we can only find some approximations to the x^y function. A technique and a hardware architecture for powering functions is presented in [3], however this implementation only deals with a previously known constant exponent. This work is extended in [4] allowing configurable exponents that can be integers or the inverse of an integer.

In this paper we present a complete powering function that allows any base or exponent in the range provided by single-precision floating-point arithmetic. Instead of using a direct approach to x^y, our unit is based on the decomposition of the powering function into a chain of operators. The main contributions of this work are the following:

- Complete implementation of single-precision f.p. x^y, with no reduction in the input ranges of the function.
- High-performance and standard single-precision f.p. implementation of the two complex operators required for computing x^y, the logarithm and the exponential function.
- Tailored implementation of the f.p. operators that allows reduced area and improved working frequency.
- Improved accuracy obtained by extending the precision of the partial results.

This paper is composed as follows: in section 2 the method employed to calculate the powering function is analyzed, while the architecture developed is exposed in section 3. Section 4 details the experimental results and finally some conclusions are drawn.

2 Powering function

The complexity of the powering function, x^y (where x is the base and y the exponent), makes very difficult to implement an efficient and accurate operator in a direct way without any range reduction. However it can be reduced to a combination of other operations and calculated straightforward with the transformation:

$$z = x^y = e^{y \times \ln x} \tag{1}$$

A direct implementation of this approach with three sub-operators (a logarithm, a multiplier and an exponential) presents three main problems that have to be effectively handled:

- The enormous complexity of both exponential and logarithm functions. However, the use of table-driven methods in combination with range reduction algorithms [5, 6] makes possible their implementation.
- The computation with a negative base results in Not a Number even though the powering function is defined for negative bases and integer exponents, see Section 2.2.
- Equation 1 can lead to a large relative error in the result. Although the sub-operators were almost exact (upto half ulp) the relative error from each sub-operator spreads through the equation generating a final large relative error [7]. Extending the precision of the partial results is an effective way to minimize these relative errors, see Section 3.2.1.

2.1 Single-Precision Floating-Point Arithmetic

In the IEEE Standard for binary f.p. arithmetic each number is composed of a sign (s), a mantissa (m) and an exponent (exp), being the value of a number:

$$s \times m' \times 2^{exp'} = s \times h.m \times 2^{exp-bias}$$

where h, also called hidden bit, can be 0 or 1 and $bias$ is a constant that depends on the bit-width of the exponent, (e), being its value $2^{e-1}-1$. The precision of the number, single or

	Input Range			Output Range		
	function	f.p.	denormalized	function	f.p.	denormalized
e^x	$(-\infty, \infty)$	(-103.98, 88.72)	No	(0,∞)	(0,∞)	Yes
$\ln x$	(0,∞)	(0,∞)	Yes	$(-\infty, \infty)$	(-103.27,88.72)	No
$\times$	($-\infty$,∞)	($-\infty$,∞)	Yes	$(-\infty, \infty)$	($-\infty$,∞)	Yes

Table 1. Sub-operators Range Analysis.

double, determines its bit-width, 32 or 64 bits. For single precision (32 bits), 1 bit corresponds to s, 8 to exp and 23 to m. With this number representation f.p. format presents up to five different types of numbers: *Normalized* (the common numbers, where h is 1), *Denormalized* (numbers very close to zero, h is 0), signed *Zeros* and *Infinities*, and *Not a Number, NaN*, (to indicate exceptions). The combination of the exponent and mantissa values determines when we are working with one type of number or another. Handling five types of numbers implies an increase of the complexity of the operators working with f.p. arithmetic because in addition to the calculation unit itself, preprocessing and postprocessing of the input and output numbers is required.

The use of denormalized numbers is responsible of most of the prenormalization and postnormalization algorithms. Generally, f.p. arithmetic operations need a normalized value format $s \times 1.m \times 2^{exp'}$ for the calculation. Denormalized numbers have to be converted to normalized ones in the preprocessing (also called prenormalization) requiring the detection of the most significant bit (msb) of the mantissa and an adjustment of the exponent and mantissa values depending of the position of the *msb*.

During postprocessing (also called postnormalization), the result of the calculation unit has to be analyzed to determine to which type of number corresponds and to make adjustments to its exponent and mantissa. Again, most postprocessing algorithms deal with denormalized numbers.

2.2 Operators Analysis

One of the key issues of the proposed x^y implementation is the analysis of the input and output ranges of the sub-operators in single-precision f.p. arithmetic. From this analysis, summarized in Table 1, we can improve the performance and accuracy of the powering function on an FPGA, see Section 3.2.1. For each sub-operator, we have analyzed the input and output ranges of the functions and the resolution allowed with single precision f.p. arithmetic, and specifically if they have resolution for denormalized numbers. The output range of the exponential function, (0,∞) is achieved with a reduced input range of (-103.98, 88.72), as it tends quickly to zero for negative inputs and to infinity for positive ones. We can also notice that for any number smaller than 2^{-24} the result is always 1. This means that the denormalized numbers, at the input, have no resolution as their result is equal to $e^0 = 1$ and therefore can be treated as a zero input. For $\ln x$, as it is the inverse function of e^x, its analysis is exactly the same as for e^x just by interchanging inputs for outputs, not generating now denormalized numbers at the output. Finally the multiplication operation has no special feature, using the complete range for both input and output including denormalized numbers.

Looking to the operation ranges from Table 1 and equation 1, it can be thought that the input ranges for x^y would be (0,∞) for the base and $(-\infty, \infty)$ for the exponent. However, the analysis of the powering function presents a variation respect the range expected from its sub-operators. The real input range of the base and exponent of the powering function includes the full range, $(-\infty, \infty)$, for both base and exponent. However, some limitations have to be introduced due to the type of results that can be obtained when that range is used.

For positive bases there is no problem and the result will be positive real numbers. But for a negative base, the common result is a complex number unless the exponent is an integer number, being the result in this case a positive real number when the exponent is even and a negative real number when it is an odd number. This circumstance can be effectively handled in a FPGA with just some extra logic, as will be explained in Section 3.2.

3 Architecture

The feasibility of our approach depends on the achievement of high-performance units for the three operators involved in the powering function. While FPGA single precision f.p. arithmetic multipliers are well known and there exist several implementations, very few implementations of both exponential and logarithm units can be found in the literature.

3.1 Exponential and Logarithm functions

To the best of our knowledge there are only two previous works focused on the exponential function [8] [9], and only one for the logarithm function [10] (from the same authors of [9]). The first one [8], employs an algorithm that does not exploit the FPGA characteristics, and consequently presents poor performance. The other two implementations [9, 10] are part of a common work and are designed suiting with FPGA flexibility (using internal tailored fixed arithmetic and exploiting the parallelism features of the FPGA) achieving much better results. They are parameterizable implementations that, additionally to single f.p. format, also allow smaller exponent and mantissa bit-widths and are both based on input range reduction and table-driven methods to calculate the function in the reduced range. Our e^x and $\ln x$ units, based on these units, include the following innovative features:

- Single precision f.p. arithmetic extension. [9, 10] were designed considering only normalized numbers, not denormalized. Additional logic has been introduced to handle denormalized numbers at the output of e^x and the input of $\ln x$.
- Redesign of units to deal only with single precision. The feature of bit-width configurability of the base designs has been removed. Thus, the resources needed have been reduced because specific units, just for single precision, have been developed.
- Simplification of constant multiplications. As suggested in [9], conventional multipliers have been removed where the multiplications involved constant coefficients, improving performance and reducing size.
- Unsigned arithmetic. In [9, 10] internal fixed arithmetic with sign is used. However, some operations (like the ones involving range reduction and calculation of the exponent for the result in e^x) are consecutive and related, and the sign of the result can be inferred from the input sign. For such operations signed arithmetic has been replaced by unsigned arithmetic with the corresponding logic reduction.
- Improved pipelining. The speed is enhanced by systematically introducing pipeline stages to the datapath of the exponential and logarithm units and their subunits.

3.1.1 Exponential function

The algorithm used to compute e^x [9] combines an approximated calculation of the exponent of the result ± 1 (k in Figure 1) with an input range reduction technique to obtain a smaller number range for the exponential computation. Applying the following expression to x the f.p. exponential function is greatly simplified:

$$x \approx k \ln 2 + y \;\rightarrow\; e^x \approx 2^k e^y$$

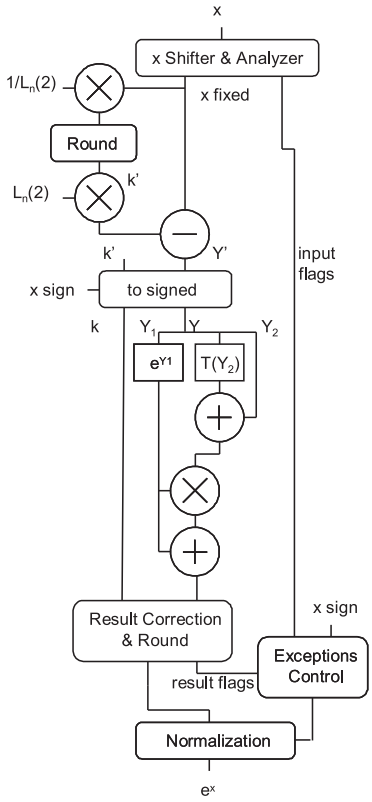

Figure 1. Exponential function unit.

being y the number that verifies that transformation. This way the computation of the exponent and the mantissa of the result is split. The used algorithm is as follows. The exponential function works internally from a fixed number of 35 bit-width, x_fixed. This bit-width is determined by the upper bound of the function (achieved with msb corresponding to 2^6 as seen in Section 2.2) and that the result of the f.p. exponential for any number smaller than 2^{-24} is always equal to 1. Due to the exponential property:

$$e^{a+b} = e^a e^b$$

the bits of the input number corresponding to an exponent smaller than -24 will have no impact on the result. This way the bit-width needed for x_fixed will be of 31 bits, but four additional bits are included as guards for the internal operations. k is calculated multiplying x_fixed by $\frac{1}{\ln 2}$ and rounding the result, while y is obtained subtracting $k \times \ln 2$ from x_fixed.

The calculation of e^y, that will generate the mantissa of the result, also involves a second range reduction as y can be split: $e^y = e^{y_1} e^{y_2}$, corresponding y_1 to the most significant bits of y and y_2 to the least significant bits. The calculation of both exponentials is based on table-driven methods being e^{y_1} calculated directly while e^{y_2} is calculated using the Taylor formula:

$$T(y_2) \approx e^{y_2} - 1 - y_2 \rightarrow e^{y_2} \approx 1 + y_2 + T(y_2)$$

$$e^y = e^{y_1} e^{y_2} = e^{y_1} + e^{y_1}(y_2 + T(y_2))$$

as can be seen in the bottom part of Figure 1. Finally the exponent of the result is adjusted depending on the mantissa value and then the postnormalization of the result is applied.

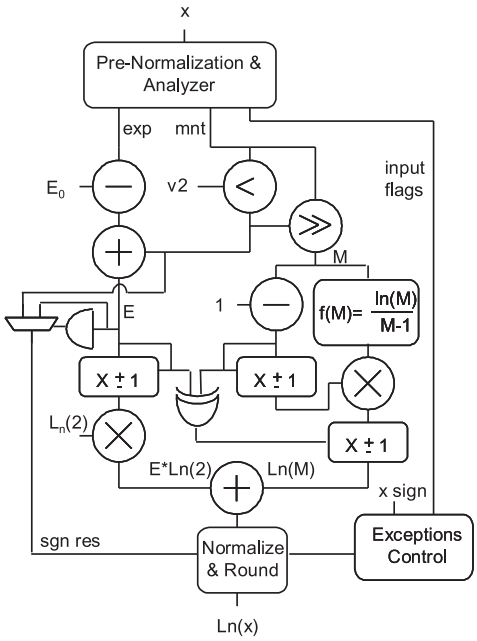

Figure 2. Logarithm unit.

3.1.2 Logarithm function

Due to the logarithm properties of $\ln(ab) = \ln(a) + \ln(b)$ and $\ln a^b = b\ln(a)$ the logarithm of a normalized number can be computed as:

$$y = \ln x \ \rightarrow y = \ln(1.mnt) + (exp - bias)\ln 2$$

During the direct computation of this formula a catastrophic cancellation of the two terms may happen. Consequently, the internal architecture of the unit will need very large bit-widths for internal variables and very large operators to maintain error bounds.

To avoid this problem, the algorithm centers the output range of the function around zero:

- $y = \ln(1.mnt) + (exp - bias)\ln 2$ when $1.mnt \in \{1, \sqrt{2})$
- $y = \ln(\frac{1.mnt}{2}) + (1 + exp - bias)\ln 2$ when $1.mnt \in \{\sqrt{2}, 2)$

where we can denote the first element as $\ln M$ and the second element (once multiplied by $ln2$) as E, see Figure 2.

This way the input range of $\ln M$ is reduced to $M \in [\frac{\sqrt{2}}{2}, \sqrt{2})$, with an output range of $[-\frac{\ln 2}{2}, \frac{\ln 2}{2})$. The calculation of $\ln M$ is done with polynomial methods and, to achieve less than one ulp error at smaller cost, $\ln M$ is not calculated in only one step but in two. When M is close to 1, $\ln M$ is close to $M-1$ so, instead of computing $\ln M$, less resources are needed if $f(M) = \frac{\ln M}{M-1}$ is calculated and then $\ln M$ reconstructed by multiplying by $M - 1$.

3.2 Powering function

As seen previously in 2.2, the x^y function requires not only three sub-operators, but also some additional hardware to efficiently handle the case of negative bases and integer exponents.

In Figure 3 the solution adopted is shown. The chain of operators is completed with an exceptions control unit, that analyzes the base and exponent to detect when the exponent is an integer (and in that case if it is odd or even to decide the sign of the result) and all the other exceptions of the powering function such as ∞^0, 0^y, 0^∞, 1^∞, NaN input, etc. Meanwhile,

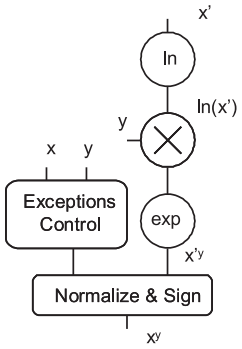

Figure 3. Powering unit.

the chain of operators will always work with a positive base x' to avoid the logarithm NaN result for negative bases. Finally the x'^y result obtained, the activated exceptions flags and the result sign calculated will be analyzed together to obtain x^y.

This way, a first implementation of the powering function will require minor changes in the three sub-operators as only some changes in the internal analysis of exceptions are required.

3.2.1 Improved Implementation

Some modifications have been introduced on the sub-operators to simultaneously obtain two objectives:

- Improving the accuracy of the result by diminishing the relative error in the partial results and therefore the global relative error.
- Diminishing the FPGA resources used by removing unnecessary logic.

A first modification is the extension of the output precision of the logarithm unit. This unit generates the mantissa of the result with an accuracy of 29 bits, the 24 required in single precision f.p. arithmetic (including the hidden bit) and 5 guard bits used to ensure a faithfully rounding in single f.p. arithmetic to the required 24 bits mantissa. However, we can eliminate the rounding, and its corresponding logic, to work in the next operator with a 29 bits mantissa instead of the standard 24 bits mantissa so the relative error associated to the logarithm function is reduced.

A second modification will be the redesign of the multiplier unit to take advantage of the type of the partial results that are produced in the modified logarithm unit (not denormalized numbers and 29 bits mantissa precision) and the numbers that are handled in the exponential unit (no resolution for denormalized numbers and conversion of input into a fixed number of 35 bit-width). As seen in Section 2.2, the logarithm unit does not generate denormalized numbers, so the logic needed to the prenormalization of that input operand can be eliminated. Additionally the multiplier has to be redesigned to work with the 29 bits mantissa input generated in the modified logarithm unit. The same circumstances happen with the result of the multiplier, as the exponential function does not have input resolution for denormalized numbers. The logic used in the multiplier for the postnormalization of denormalized results can be also eliminated handling them as zeros. Additionally, the result generated by the multiplier, before rounding, will have a mantissa bit-with of 52 bits plus a carry bit with an accuracy of half ulp, while the exponential unit works transforming its input into a 35 bit-width fixed number. So, the rounding stage can be eliminated and we can directly use a multiplier output of 35 bits mantissa precision.

	Slices	18x18 Mult.	Block RAMs	Clock [MHz]	Stages
[9]	1037	0	0	149.0	19
Standard v1	1285	0	0	250.2	19
Standard v2	543	4	1	259.9	17
Improved x^y	577	4	1	252.1	17

Table 2. Exponential Operator.

	Slices	18x18 Mult.	Block Rams	Clock [MHz]	Stages
[10]	1090	0	3	172.8	13
Standard v1	1361	0	3	233.2	21
Standard v2	933	5	2	241.6	18
Improved x^y	848	5	2	241.1	16

Table 3. Logarithm Operator.

Consequently, the last modification to achieve the aforementioned objectives will be the adaptation of the exponential unit to the new inputs with 35 bits mantissa. This affects to the conversion of the input into the internal fixed number required. In the standard operator, although the fixed number obtained has a 35 bit-width it was limited to a real resolution of up to the 24 bits of the mantissa. Now, as the mantissa has a 35 bits resolution the fixed number can also have up to 35 bits resolution. In this way the relative error of the exponential and multiplication units is reduced and therefore the global relative error.

4 Experimental Results

The architecture of the powering unit and its sub-operators has been implemented on a Xilinx Virtex-4 XC4VF140-11 FPGA. The results are divided into two main parts. Firstly, the results for the sub-operators for both a direct implementation of x^y or the improved architecture are described. Also the exponential and the logarithm operators are compared with the base implementations [9, 10]. Secondly, the results for the complete x^y are presented.

4.1 Sub-operators: Exponential function, Logarithm and Multiplier

Table 2 summarizes the results obtained for the exponential operator for four different implementations we have carried out. The first three implementations correspond to single precision f.p. arithmetic units, a pipelined version of the selected base implementation [9] and our redesigned units with two configurations. To make a fair comparison with [9], standard v1 uses the same type of FPGA resources as [9], while standard v2, is the optimal configuration using embedded multipliers, block RAMS and KCM multipliers. The fourth implementation corresponds to our exponential unit modified for the improved version of x^y.

Comparing [9] with v1, it can be observed that the speed of the unit has been substantially increased by a 67.8% without having increased the number of stages of the pipelined architecture while the logic used is only 23.9% bigger. This increase is due to the redesign of the datapath and the multipliers (including new pipeline stages and the use of faster but more expensive architectures) and the implementation of the logic needed for handling denormalized numbers that were not considered in [9]. On the other hand, the use of unsigned arithmetic for consecutive operations instead of signed arithmetic has eliminated unnecessary stages and logic.

For the optimal implementation, v2, introducing embedded multipliers to replace the biggest LUT based multiplier in the unit (which is responsible for almost half of the unit area, 592 slices), KCM multipliers where possible and a block RAM for the largest LUT exponent

	Slices	18x18 Mult.	Block Rams	Clock [MHz]	Stages
Standard	484	4	0	213.4	8
Improved x^y	250	4	0	224.7	6

Table 4. Multiplier.

	Slices	18x18 Mult.	Block Rams	Clock [MHz]	Stages
Direct	2019	13	3	201.3	43
Improved	1732	13	3	201.3	37

Table 5. Powering Unit.

table [9] reduces significatively the logic needed, in a 47.6% with respect to the base implementation and 57.7% with respect to v1. Moreover, two pipeline stages are eliminated replacing the LUT based multiplier while speed is increased in another 3.8%. Finally, the improved implementation (the adaptation of v2 to the input mantissas of 35 bits precision) means an increase of a 6.2% of logic used, as the shifter unit in charge of transforming the f.p. number into a fixed one has to work with a bigger mantissa, and a loss of a 2.6% of speed.

The same comparison, with the same four types of implementations, has been realized for the logarithm. Now, as can be observed in Table 3, improving the speed of the unit by a 34.9%, has much more impact on the resources used and the number of pipeline stages, using v1 24.9% more logic than [10] and requiring eight more stages.

For our optimal configuration, v2, now two LUT multipliers are replaced using five embedded multipliers while a coefficient table is implemented as a LUT table instead of using a Block RAM as its size is very small and it requires few resources. Now the reduction achived in the resources used has been of 14.4% with respect to [10] and 31.4% with respect to v1, while three stages are removed in the multipliers and also obtaining an additional increase of 3.6% in the speed. And finally, the version of the logarithm operator for the improved x^y, means another logic reduction of 9.1% and the reduction of two stages as the rounding logic has been removed.

Table 4 compares our standard implementation of a single-precision f.p. multiplier with the modified implementation used for the improved x^y. The elimination of the logic needed for denormalized numbers (at the multiplier output and the input corresponding to the logarithm output) and for rounding, improves the performance of the multiplier, reducing a 48.3% the logic used and two stages of the pipeline, while the working frequency is speeded up a 5.3%.

4.2 Powering function

Finally, the global results for our powering unit are summarized in Table 5 for two possible implementations. The first one, named as *direct*, would be the equivalent to a single precision software x^y calculated straightforward from equation (1) with the modifications needed to correctly compute negative bases. This way it is composed of the optimal standard implementations of the sub-operators (v2) and the logic required for exceptions.

On the other hand, the improved implementation includes the proposed modifications to reduce the relative error and also improve the performance, and is composed of all the modified sub-operators. As can be observed, in addition to reduce the relative error of the result, the logic needed for the improved implementation, is 14.2% smaller while the pipeline architecture is reduced in 6 stages. Considering the sub-operators individually this reduction should be of only four stages (two from the logarithm and two from the multiplier). An additional reduction of two stages is obtained parallelizing the last stages of the logarithm with the multiplier prenormalization logic.

Finally, even though the FPGA used is one of the largest of the family we can remark that the resources needed for our improved x^y represent just a 2% of the logic, 6% of the embedded multipliers and 0.5% of the block RAMs. Therefore, plenty of resources are left available for the implementation of other logic where the x^y can be integrated.

5 Conclusions

This work has presented an FPGA implementation of the powering function with single f.p. arithmetic. Our implementation computes x^y as a chain of sub-operators, a logarithm, a multiplier and an exponential function. This approach allows the hardware implementation of x^y without limiting the input range of the base and exponent of the operation while the introduction of additional logic handles the cases where x^y is defined for a negative base. High performance single f.p. sub-operators have been designed and modified to achieve an improved x^y with the goal of reducing the relative error associated to the approach used. Our FPGA operator achieves high performance due to a pipelined architecture with a 201.3 MHz clock and a throughput of one sample per cycle on a Xilinx Virtex-4 FPGA.

References

[1] G. Govindu, R. Scrofano and V. K. Prassana, "A library of parametereizable floating-point cores for FPGAs and their application to scientific computing," in *The International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2005, pp. 137–148.

[2] X. Wang, S. Braganza and M. Leeser, "Advanced components in the variable precision floating-point library," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.

[3] J. A. Pieiro, J. D. Brugera and J. M. Muller, "FPGA implementation of a faithful polynomial approximation for powering function computation," in *Euromicro Symposium on Digital System Design*, 2001, pp. 262–269.

[4] J. A. Pieiro, M. D. Ercegovac and J. D. Bruguera, "Algorithm and architecture for logarithm, exponential and powering computation," *IEEE Transactions on Computers*, vol. 53, no. 9, pp. 1085–1096, 2004.

[5] P. Tang, "Table-driven implementation of the exponential function in IEEE floating-point arithmetic," *ACM Transactions on Mathematical Software*, vol. 15, no. 2, pp. 144–157, 1989.

[6] ——, "Table-driven implementation of the logarithm function in IEEE floating-point arithmetic," *ACM Transactions on Mathematical Software*, vol. 16, no. 4, pp. 378–400, 1990.

[7] J. Muller, *Elementary Functions. Algorithms and Implementation.*, Birkhauser, Ed., 1997.

[8] C. C. Doss and R. L. Riley, "FPGA-Based implementation of a robust IEEE-754 exponential unit," in *IEEE Field-Programmable Custom Computing Machines*, 2004, pp. 229–238.

[9] J. Detrey and F. de Dinechin, "A parameterized floating-point exponential function for FPGAs," in *IEEE International Conference Field-Programmable Technology*, 2005, pp. 27–34.

[10] ——, "A parameterized floating-point logarithm operator for FPGAs," in *Signals, Systems and Computers, 2005. Conference Record of the Thirty-Ninth Asilomar Conference*, 2005, pp. 1186–1190.